A Testability-based Assertion Placement Tool for Object-Oriented Software

National Institute of Standards and Technology
SBIR Phase II Final Report
for Contract Number: 50-DKNB-4-00185

by

J. Voas, M. Schatz, M. Schmid
Reliable Software Technologies Corporation
Loudoun Tech Center
Suite 250, 21515 Ridgetop Circle
Sterling, Virginia 20166
http://www.rstcorp.com

September 11, 1997

This material is based upon work supported by the Deparment of Commerce under contract number 50-DKNB-4-00185. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the Department of Commerce.

Contents

1 Executive Project Summary **2**

2 Introduction **3**

3 Run-time Software Assertions **4**

3.1 Strategic Assertion Placement 8

3.2 Generalized Observations Concerning Assertions 9

3.2.1 Reachability 9

3.2.2 Changing Test Suites 10

3.2.3 Testers vs. Developers 11

4 Our Phase II Prototype Tools **13**

4.1 Components 13

4.2 Styles of Assertions 15

4.3 Types of Assertions 15

4.3.1 The ASSERT statement 15

4.3.2 The PRE_CONDITION and POST_CONDITION Statements 16

4.3.3 The INVARIANT Statement 17

4.4 The Firing of Assertions 18

4.5 Assertion Behaviors 20

4.6 The Assertion Language 21

5 Using Testability to Guide Assertion Placement **23**

6 Summary **25**

List of Figures

1 One-dimensional output. 6
2 Two-dimensional output from added Assertion 7
3 Assertions increase the cardinality of the output space. 8
4 The Testability-guided Assertion Placement Process. 14
5 Assert Wizard for Placement. 24
6 A.E.W. 25

Abstract

Dynamically testing software that has been augmented with assertions increases the defect observability of the test cases (provided that the assertions are reached during testing). This report presents an approach and tool for assertion localization that is based on finding regions of the code that appear to be untestable, and then making them more testable. In our model, assertion localization is accomplished by using dynamic testability analysis. Our testability analysis is implemented using fault injection, and it suggests where faults can hide during testing. This report also explores the phenomenon where assertions that were designed to boost the fault observability provided by test scheme D also boost the fault observability afforded by a different testing scheme, D'. This phenomenon demonstrates a unique and cost-effective benefit of assertions not before exploited, and lays forth a new avenue for finding higher return-on-investment testing techniques.

Keywords

Assertions, faults, failure, testing, fault detectability, oracles, specifications.

1 Executive Project Summary

This document is the Final Report for SBIR contract 50-DKNB-4-00185, titled "Assessing and Increasing the Testability of Object-Oriented Systems" for the National Institute of Standards and Technology (NIST). In this Phase II effort, Reliable Software Technologies (Sterling, VA) has developed the following software utilities: (1) an assertion placement tool for C++ software programs, (2) an assertion language tool for C++, and (3) an assertion language tool for Java. Contractually, Reliable Software Technologies was required only to develop the first and second of these for NIST, but because of the commercialization expectations from Phase III, Java provides a much stronger business opportunity for Reliable Software Technologies. Thus we added the Java assertion language capability, and are now ready to begin the Phase III process.

The C++ utilities allow developers and testers to more easily embed assertions based on the recommendations that come after performing software sensitivity analysis [8] on a program's source-code. Sensitivity analysis is a source-code–based fault injection utility that recommends where assertions are warranted. These recommendations are based on the results from running the software with test cases and simulated faults that are instrumented in the software. If a suite of test cases cannot detect simulated faults, then that suite is less likely to detect real faults. This isolates those places in the code that are good candidates for receiving assertions.

Because we have not yet completed the development of our Java sensitivity analysis tool (which is outside of the scope of this *Statement of Work*), the Java assertion language tool only helps a developer instrument code with assertions, and does not recommend where to place them as the C++ tool does. The Java sensitivity analysis tool is under development, however, and is anticipated for release in 1998.

Run-time software assertions are merely "little tests" that are themselves code that are embedded in software to check to see that certain states in an executing program have certain qualities. The typical goal of using assertions is to show that program states are correct, and since state correctness is defined by the specification, assertions can test that various requirements of the specification are correctly implemented.

This two-year effort started in October, 1995, and was completed in late September, 1997. This report contains an overview of the theory that is used to determine where assertions are needed and what those assertions need to check for. This report also describes the architecture of the tools. Details that pertain to using the tools are not captured in this report, but are provided in the *User's manual* that accompanies the tools.

2 Introduction

Software testing is generally performed for one of two reasons: (1) to detect the existence of defects (*faults*), and (2) to estimate the reliability of the code. Although there are other uses for software testing, most applications of it can be grouped into one of these two categories. Errors are mental mistakes made by programmers, and faults are the manifestation of those errors in the code.

Testing is considered effective when it uncovers defects. Testing will often be considered ineffective when no failures occur, because few people today consider defect-free code to be an achievable goal. Residual software defects not discovered during testing can have significant and dangerous consequences after the software is released. Since debugging is usually invoked only after software failure is observed, test schemes that provide a greater ability to cause software failure if defects are present are very desirable. Using software assertions in conjuction with testing is one way to make software faults more visible.

Software testability is a characteristic that either suggests how easy software is to test, how well the tests are able to interact with the code to reveal defects, or some combination of the two [8]. Describing how easy software will be to test is valuable information for project schedules and project cost estimation, but this information provides little insight into how well your test case generator is at creating "defect-detecting" test cases. Because of this deficiency, it is useful to define software testability as a measure of how well test cases make defects *detectable*. This will be the perspective taken throughout this report.

Using this definition of software testability, when software is assessed as having higher testability, it means that incorrect output will likely occur if a defect exists.[1] To understand why faults hide during testing, it is necessary to know the sequence of events that must occur in order to observe incorrect output:

1. An input must cause a defect to be *executed* (or what is sometimes referred to as "reached").
2. Once the defect is executed, the succeeding data state must become corrupted. This data state is hence referred to as containing a *data state error*.
3. After a data state error is created, the data state error must *propagate* to an output state, meaning that incorrect output exits the software.

This sequence of events is called the "fault/failure" model, because it relates faults, data state errors, and failures [8]. Since faults trigger data state errors that in turn trigger software

[1]Testability scores are in the range [0, 1], with 1 being the highest and 0 being the lowest. This is because they are probabilities.

failures, any analysis that claims to suggest whether testing is capable of detecting defects must account for all three conditions[2].

This report argues that by using heuristics that identify regions in which faults cannot be detected (by testing alone, i.e., without assertions), we can identify *where* additional validation efforts (which may include more testing or non-testing approaches) should be performed. We provide a methodology that is complementary to traditional testing and that reduces the possibility of defect-hiding in those code regions. Our approach can be thought of as a strategic assertion placement heuristic. Using this technique improves the likelihood of defect detection and hence improves the software's reliability.

Over the years, we have observed that although testers would like to use assertions to improve the quality of their testing process, they find that they do not know the code well enough to inject correct assertions. For this reason, assertions are primarily used by developers. However when developers are responsible for creating assertions, this increases the likelihood that if the code is wrong, the assertions that are derived will be wrong. Further, neither group really knows where to place assertions that are the most cost-effective. All of these problems should not be disheartening, because as you will see, a handful of correct and strategically-placed assertions can dramatically improve the quality of a finished software product.

Unfortunately, this report does not offer foolproof solutions to these problems, but it does provide additional reasons for why assertions must be used to improve the deficiencies of software testing (even if that requires deriving assertions from formal specifications). Interest in assertions has in fact become so great that several recent languages support assertion placement, including Anna [5] and Eiffel [7].[3] This report suggests how developers and testers can form a partnership that, if successful, can begin to alleviate several of the aforementioned problems.

3 Run-time Software Assertions

Manually finding defects in a program's output is difficult if failures occur rarely. Manual debugging is simply a task that humans are not proficient at. Hence, the use of *automated testing oracles* (or what throughout this report we simply term "oracles") is valuable when testing software in which failures are rare. The software is of good enough quality such that failures are rare.

Building automated test oracles requires that the oracle knows what is correct (with respect to the specification) and what is not. Fortunately, oracles can be designed directly

[2]Faults and defects may be used interchangeably as they are synonymous terms.

[3]Anna (Annotated Ada) uses comments to embed assertions; Eiffel uses object invariants that are inserted as pre- and post-conditions to all operations on the object.

from *formal specifications* (that describe exactly what the software is supposed to do without also describing the implementation details of the system) [11]. An *executable specification* is a formal specification can be executed (like a program) to see if the behavior it defines satisfies the higher level requirements of the system. Executable specifications typically produce output, but do not check this output. In theory, the output from an executable specification can be given to the oracle so that the oracle can learn what output is correct.

Run-time assertion checking is a "programming for better validation" trick that helps ensure that a program state satisfies certain logical constraints. Unlike executable specifications, run-time assertions do check for the correctness of the output. *Run-time assertions* (or simply "assertions") are based on either the requirements or the specification. Some of the earliest writing concerning assertions can be traced to [6, 9], and more recent research into giving programs the ability to check themselves during execution can be found in [12, 13, 2, 15]. Further, Bieman and Yin recently discussed using assertions to increase fault detectability [1]; however our contribution furthers the idea of run-time assertions by providing methods for placing assertions where assertions are more desperately needed. Our method decides *what portion* of the software's state really needs to be checked, and *where* that check needs to be placed. We term our assertion placement scheme *strategic run-time assertion checking.*

Our goal is to embed assertions in a manner that engenders testing with greater defect revealing ability. The conjecture that has motivated this work follows:

> Why place assertions on program states if it is known *a priori* that *if* these states are in error, failure of the software is nearly guaranteed to occur. Instead, place assertions on program states when it is likely that incorrectness in those portions of the state will not be observable in the software's output.

Our strategic run-time assertion checking presents the opportunity to thwart defect hiding at a more reasonable cost than *ad hoc* assertion placement, which is the usual heuristic for deciding where to put assertions.

Software assertions can be very complicated in terms of the information that they produce, but for simplicity, we will assume that software assertions are Boolean functions that only evaluate to TRUE when a program state satisfies some semantic condition, and FALSE otherwise. If an assertion evaluates to FALSE, we will consider it the same as if the execution of the program resulted in failure, even if the output for that execution is correct. Because a program that did not have assertions is truly a different program after assertions are added, it is necessary for us to redefine what we consider a program failure: a program *failure* will be said to have occurred if the program output is incorrect *or* if an assertion evaluates to FALSE. This not only modifies what is considered failure, but it also modifies what is considered output, because there is now one more bit of output each time that an assertion statement fails.

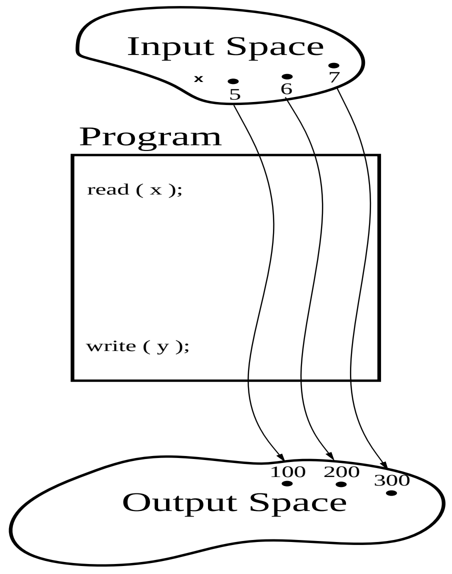
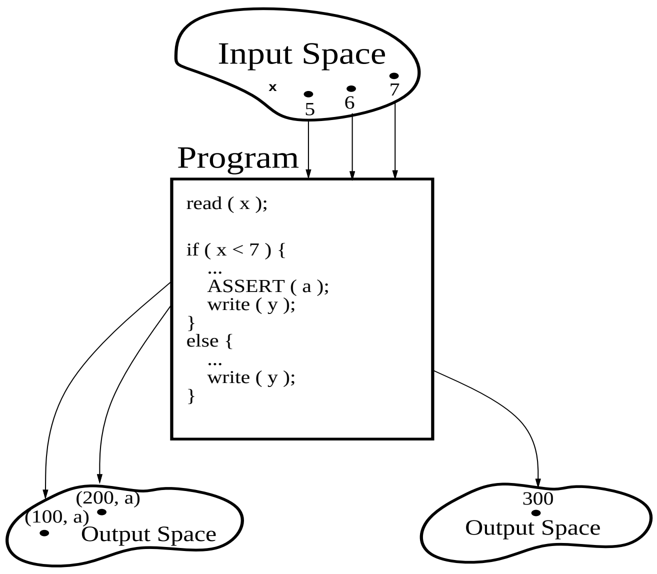


Figure 1: One-dimensional output.

Whenever the amount of output increases (*e.g.*, outputting 2 64-bit floating point values as opposed to one), more of the internal (intermediate) calculations can be *observed*. *Observability* has long been a metric used in hardware design that describes the degree (or ability) of a chip to detect problems in the inner logic of a chip at the output of the chip. When observability is poor, hardware probes are often placed in circuits to increase the observability of the circuit during testing. Similarly, assertions increase software's observability by increasing the dimensionality and/or cardinality of the software's output space, which is precisely what is desired if the of testing is to catch defects. (This is discussed in more detail later.) In our work, we consider software observability to be an informal measure of the tester's ability to ascertain what is occurring internally inside of the states created at run-time.

Correctness proofs can be thought of as a formal assertion checking system. Correctness proofs statically test to ensure that the entire program satisfies certain logical constraints for all inputs, whereas an executable specification, like a program, can be run on a per test case basis. Software assertions perform a slightly different function than correctness proofs; they semantically test internal program states that are created at run-time and that may not be observable as stand-alone entities. For example, given a known range of legal values for some intermediate computation in a program, a software assertion can test the correctness of the program state at the instant when the state is created. Since assertions are able to check intermediate data state values, they can reveal when the program has entered into an undesirable state. This is vital, because the undesirable state might not propagate into a program failure.



```
read ( x );

if ( x < 7 ) {
   ...
   ASSERT ( a );
   write ( y );
}
else {
   ...
   write ( y );
}
```

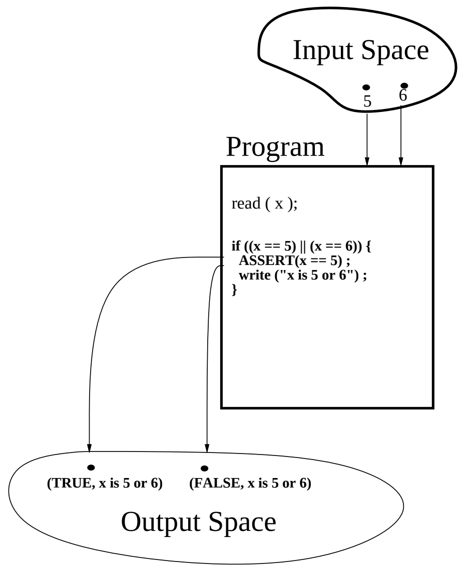
Figure 2: Two-dimensional output from added Assertion

An important measure of the fault-hiding ability of a program is contained in the dimensionality and cardinality of the program. Dimensionality is the number of different output variables produced by the program, and cardinality is the total number of unique output values. So for example, if a program has 100 different variables, and only one variable's value is output, then the dimensionality is one. If 20 variables have their values output, the dimensionality would be 20. If there are 1,000,000 million unique input values to the program, then the cardinality of the program is less than or equal to 1,000,000.

The effect of assertions on the dimensionality and cardinality of a program can be best explained through examples. Figure 1 illustrates a program that reads in an integer and outputs an integer; in this example, an input value of 5 produces 100, 6 produces 200, and 7 produces 300. Thus, the dimensionality of the output space in Figure 1 is one.

In Figure 2, the conditional branch in the code causes only certain inputs to execute the assertion. The assertion essentially acts as another output statement whose result will be checked by the oracle. Thus, for some inputs, the dimensionality of the output actually increases to two. In Figure 2, the inputs 5 and 6 execute the assertion and will therefore have outputs with a dimensionality of two, whereas an input value of 7 will have a dimensionality of one.

Now imagine a slightly different example where two unique input cases resulted in the same output value, and assume this value is of dimension n. By adding an assertion to the code that both input cases execute, it is possible that the variable asserted on now has different values, and hence each input case can be thought of as producing a unique output value of dimension $n+1$. This is shown in Figure 3. In this example, we see how an assertion



```
read ( x );

if ((x == 5) || (x == 6)) {
  ASSERT(x == 5) ;
  write ("x is 5 or 6") ;
}
```

Figure 3: Assertions increase the cardinality of the output space.

can increase the cardinality of the output space.

Assertions have an interesting ability that is similar to the ability of oracles. When an assertion evaluates to FALSE, it has the ability to not only warn of a problem at this point in the code, but also to warn of problems at totally different places in the code. This requires reversing the execution trace back to the preceding computations in order to localize the problem. It is this ability to warn of problems originating from various statements that increases the fault detectability provided by assertions.

3.1 Strategic Assertion Placement

We advocate a middle ground between no software assertions at all (a common practice that fortunately is becoming less common) and the maximum of assertions on every statement in a program. Our compromise is to place assertions *only* at locations where traditional testing is unlikely to uncover software defects. Once testing is completed, the embedded assertions may be removed or deactivated.[4]

Assertions can be extremely powerful testing aides; however they are expensive to derive, instrument (insert), and execute. Thus if we can find ways to better isolate where they are needed, we can not only improve the likelihood of fault detection at those places, but we can also avoid the cost of using assertions at those places where they are less helpful.

To date, assertion localization has been, at best, performed in an adhoc manner. The

[4]The removal of assertions is motivated by the desire for more efficient execution during production runs.

process has traditionally been developers sprinkling code with assertions in places that interested them. Our model is that assertions need to be placed where the test cases that will be used on the code are helpless in detecting faults. In our opinion, this makes assertion placement more systematic, and enhances the ability of software testing to detect faults.

Predicting where faults may hide is an expensive process because there are so many considerations that must be factored in. We employ sensitivity analysis to gather this information. Sensitivity analysis is a dynamic approach for predicting where faults will hide from test cases [8]. Sensitivity analysis predicts the likelihood that a test scheme will:

1. exercise the code (i.e., enforce *reachability*),
2. cause internal states to become corrupted when defects are exercised, and
3. propagate data state errors to the output space.

These three conditions must occur in order for defects to be observed. To assess reachability, sensitivity analysis tracks how frequently statements in a program are exercised. To assess (2) and (3), sensitivity analysis employs a variety of different *fault-injection* techniques that both mutate the software and mutate the internal program states that are created during execution.

3.2 Generalized Observations Concerning Assertions

To close out the theoretical part of this report, we wish to put forth several observations that we have discovered regarding the relationships between assertions, test cases, and a program's ability to hide faults. Interest in using assertions continues to grow, but many practitioners are not sure how to begin using assertions or how to convince their management that assertions should become a standard part of their testing process. Hopefully, some of these observations will provide the insights as to what assertions and reachability analysis can do towards making validation efforts more fruitful.

3.2.1 Reachability

Static metrics cannot as easily nor accurately address software reachability as can dynamic metrics. Although collecting data about information loss is useful for assertion placement, reachability analysis is still necessary to determine whether the assertion will be exercised.

Let $T(P)_D$ represent the fault detectability of program P when tested with test suite D. And let $(D \cup \Delta)$ represent test suite D after it has been augmented with enough test cases such that all statements in P are exercised.[5] Note that Δ could be empty, in which case $D = (D \cup \Delta)$. Since reachability is the first event in the fault/failure model, it is a necessary condition for faults to be detected, hence

$$T(P)_D \leq T(P)_{(D \cup \Delta)}.$$

(This is similar to Weyuker's Monotonicity Axiom [14].)

3.2.2 Changing Test Suites

Are faults more or less likely to be hiding after assertions are added and the manner by which test cases are generated changes? We begin to answer this question by looking at the situation where the code has no assertions, and then we will consider the situation where assertions are embedded.

For program P without assertions, we showed that regardless of what Δ added to test suite D, we know that $T(P)_D \leq T(P)_{(D \cup \Delta)}$. Given a new test suite D' that is not identical to D and neither test suite is a subset of the other, what can we say about the relation between $T(P)_D$ and $T(P)_{D'}$? Quite simply, we cannot say anything confidently without running a technique similar to sensitivity analysis with both D and D'. That is, we do not know whether $T(P)_D \leq T(P)_{D'}$ or $T(P)_D > T(P)_{D'}$.

Now let P_{A_D} represent program P with assertions which were designed to boost the fault revealing ability of D. Assertions increase the likelihood that the third condition of the fault/failure model happens. Then,

$$T(P)_D \leq T(P_{A_D})_D.$$

From there, this is also true:

$$T(P)_{D'} \leq T(P_{A_D})_{D'}, \tag{1}$$

regardless if D and D' have common members. Our argument in support of this conjecture follows: if the assertions placed into P (that are based on D) are never exercised via inputs from D', then $T(P)_{D'} = T(P_{A_D})_{D'}$; if as little as one assertion is exercised by some input in D', then it is possible that $T(P)_{D'} \leq T(P_{A_D})_{D'}$ will be true.

What this suggests for assertions is straightforward. Since assertions directly affect propagation, if propagation is homogeneous for some $i \in D$, it is likely to be homogeneous for some other $j \in D$.[6] Whether j is in D or D' is immaterial. In general then, assertions appear

[5]There are software test case generation tools (Godzilla [10], WhiteBox (TM) TGen) that are intelligent enough to sometimes find test cases that will exercise previously unreached statements in the code, but this is an unsolvable problem in general.

[6]Homogeneous propagation is said to occur for some input i if any type of corruption of the state created at location l by input i results in corrupted output.

to have an impact on fault detectability that is without respect for whether the inputs to the software are from one test suite or another. Hence since assertions improve the propagation prospects for inputs from D by increasing the cardinality or dimensionality of the output space, they should also improve the propagation prospects for D'.

And finally, we suspect that it will generally be true that:

$$T(P)_D < T(P_{A_D})_{(D \,\cup\, \Delta)} \tag{2}$$

However there will be cases where

$$T(P)_D = T(P_{A_D})_{(D \,\cup\, \Delta)},$$

specifically when $D = (D \,\cup\, \Delta)$ and/or the assertions placed into P are not exercised.

3.2.3 Testers vs. Developers

Assertions can be derived as soon as a specification exists, however in this code-based methodology, assertion localization does not occur until after code testability is measured. Thus since testability measurements must be performed before the assertions are derived, this process cannot be performed until later in the software development life-cycle.

However, knowing that, for example, variable **a** needs to be tested after assignment statement 100, does not indicate what constraint the assertion should be testing for (to determine whether a problem with **a** has occurred). Hence our methodology has only addressed one part of the oracle/assertion problem, specifically *placement*, not *derivation*. Someday we would like to link this tool up with a formal specification language, and automatically generate the assertions, but that remains a research goal for the future.

As mentioned earlier, testers are likely to be incapable of deriving correct assertions, while developers are more likely to derive assertions that mimic the semantics of the code already there. If the developer does not understand the specification, their assertions will be incorrect. That is to say that if a developer's code is faulty, their assertions will almost certainly be faulty.

This brings us to our final recommendation for how to team developers and testers in a partnership to strategically derive and embed assertions. Let the testers find where assertions are needed, and leave it to the developers to determine what the assertions should be. Note that this is different than the case where the developer determines both where to place the assertions and what they should be. Here, the developer is being forced to derive assertions that the tester needs for places in the code *where the developer might not be as sure as to what the assertion should be.* If this happens, it forces developers to dig deeper into the code and requirements than they might have already done. This plays a similar role as code inspections, except that the person digging into the code is also the person that is likely to have written the code. Although this is not a foolproof solution, it is the best recommendation that we can provide at this time. Having developers spend more time comparing their previous understanding of the code to the existing requirements can only serve to improve the code's quality. Even if the assertion that the developer derives does not detect an error, the fact the the developer is forced to derive the assertion will increase the likelihood that the developers themselves find errors, because they are forced to get more familiar with less familiar internal computations in the code.

4 Our Phase II Prototype Tools

The tool that we are delivering to NIST is named ASSERT++. ASSERT++ has two sub-utilities:

1. A utility that supports the assertion language that we have created. This works for both Java and C++.

2. A utility that we call `CBrowse`, which recommends where assertions should be placed based on the testability scores.

These tools are quite simple to use, and are well suited for the tester/developer team that we have advocated.

The first tool currently works for Solaris but could easily be ported to other UNIX platforms like SunOS, HPUX, and Linux. Also, this tool could be ported to Windows-NT or Windows95 quite easily. The CBrowse tool that does the testability-based placement only works on the UNIX (currently Solaris) platforms.

The first utility allows a developer to include complex assertion statements directly in the source code of a program that he or she is working with. The assertion statements have the appearance of C++ / Java statements, and are included in the source code just like any other programming statement. These statements are transformed into valid C++ / Java statements by this utility, and can then be evaluated at run-time. This utility is composed of three parts: (1) the shell, (2) the instrumentor, and (3) the compiler. These components work together to allow the user to place complex assertions in his or her code, and to have these assertions evaluated at run time.

The second utility, `CBrowse`, is the tool that allows the user to see his or her code with the testability scores, thus suggesting where assertions are needed. For this tool to work properly, it must have access to the source code as well as to the testability scores after sensitivity analysis is performed.

Figure 4 shows how this Phase II innovation fits in the software Validation and Verification process. Here, we see that the original source-code first receives software testability analysis. From that, the tester looks at the results and makes recommendations to the developer as to where assertions should be placed. The developer then derives the appropriate predicates, and the assertion instrumentor then adds the assertions to the source code, thus creating the annotated source-code version. Now, this new version of the code is ready for testing.

4.1 Components

RST's shell program is used to parse command line input. It begins by determining what kind of program it is dealing with (C++ / Java). It then breaks the compilation process

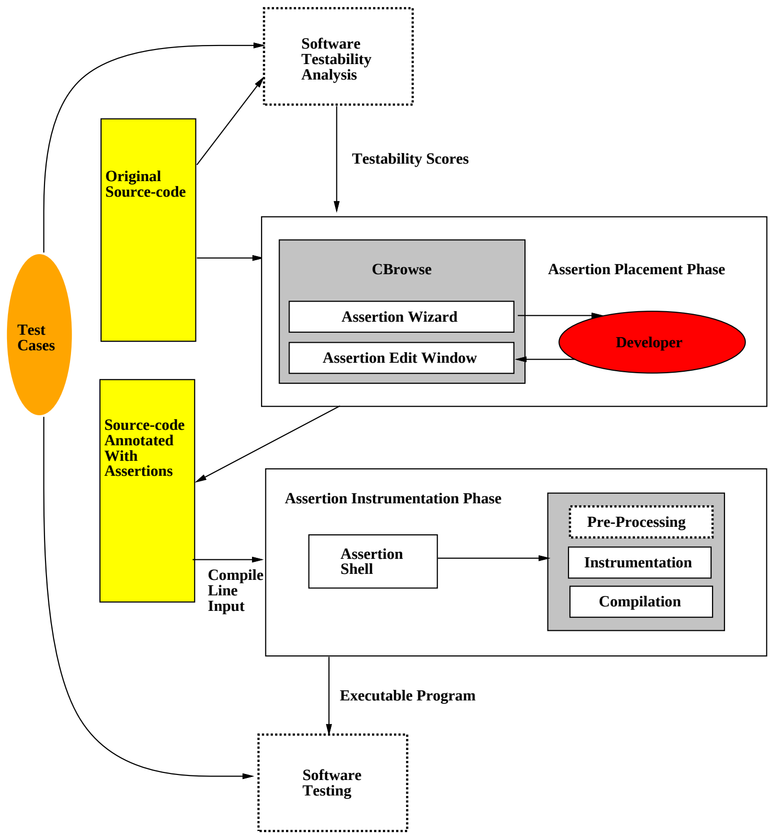


Figure 4: The Testability-guided Assertion Placement Process.

into a number of steps depending on the program type. For a C++ program, the shell determines what compiler options belong to the pre-processor, what belong to the compiler, and what belong to the linker. In the case of a Java program, the shell does not have to do any pre-processing.

The shell program then begins to execute the steps that it has determined to be necessary. For a C++ program, this means that it must first pre-process the source code. A Java program will skip this step, and move immediately to the next step, instrumentation. After invoking the correct instrumentation program (C++ / Java), the shell program will perform any necessary compilation and linking.

RST's assertion instrumentation program is responsible for parsing the source code of a program and generating instrumented source code. It translates assertion language statements into valid C++ / Java statements, which are then written into the instrumented source code. The original source code (complete with assertion statements) is never changed, and it is the instrumented source code that the assertion tool will compile.

The C++ version of RST's assertion tool can be configured to work with almost any command-line C++ compiler that can compile ANSI C++ statements. This means that a developer will most likely not have to change compilers in order to use this tool. The Java version of RST's assertion tool makes use of any command-line Java compiler. As with the C++ version of the tool, this means that in most cases the developer will not need to change compilers in order to use RST's assertion tool.

4.2 Styles of Assertions

RST's C++ assertion tool supports two styles of assertion statements. These styles are Definition Placed assertion statements and Specification Placed assertion statements. A Definition Placed assertion statement is located in the actual method/function definition. A Specification Placed assertion statement is located in the class specification section of a C++ program. RST's Java assertion tool supports only Definition Placed assertion statements since there is no class specification section of a Java program. Both the C++ assertion tool and the Java assertion tool use the same format for their assertion statements. This format is:

```
<assertion type> (<condition>, <message>) ;
```

The $< assertiontype >$ can be either ASSERT, PRE_CONDITION, POST_CONDITION, or INVARIANT. The $< condition >$ is a valid boolean statement written using the assertion language that is part of RST's assertion tool (that is described in the *User's Manual* that accompanies the ASSERT++ tool). The $< message >$ is a text string that will be displayed when the $< condition >$ evaluates to FALSE (referred to from here on as an assertion firing).

4.3 Types of Assertions

4.3.1 The ASSERT statement

An ASSERT statement is used to check the value of a single variable at a particular location in the program. It may be placed at any point that a C++ / Java statement could be placed. In the following example, an ASSERT statement is used to insure that the variable x does not have a negative value after the assignment statement.

```
. . . .
x = y ;
ASSERT( x >= 0, "x has a negative value") ;
cout << "Value of x is " << x << endl ;
. . . .
```

ASSERT statements are useful to verify that values computed within methods/functions are within acceptable bounds. An ASSERT statement may only be used as a Definition Placed assertion.

4.3.2 The PRE_CONDITION and POST_CONDITION Statements

The PRE_CONDITION and POST_CONDITION assertion statements are modeled after the ASSERT statement provided in the C language. They are used to check the value of variables at the point of method/function entry and exit respectively. A PRE_CONDITION is evaluated as the first statement of a method/function. A POST_CONDITION is evaluated as the last statement of a function/method before the return statement (or end of the function, if no return statement is provided). If multiple return statements exist, then the POST_CONDITION will be evaluated before whichever return statement is reached.

Both PRE_CONDITION and POST_CONDITION statements may be used as either Definition Placed assertions or Specification Placed assertions. When used as a Definition Placed assertion, a PRE_CONDITION or POST_CONDITION should be placed immediately after the opening "curly brace" of a function/method. POST_CONDITIONs will be evaluated at the proper place in the function/method despite their location at the top of the function / method definition. An example of a Definition Placed PRE_CONDITION and POST_CONDITION is:

```
void
Screen::LightPixel(int x, int y)
{
  PRE_CONDITION ((x >= 0) && (y >= 0)), "Negative screen coordinate given") ;
  POST_CONDITION( screen[x][y] == true, "Failed to light pixel") ;

  screen[x][y] = true ;
  return ;
}
```

The above example is equivalent to writing the following:

```
void
Screen::LightPixel(int x, int y)
{
  ASSERT((x >= 0) && (y >= 0)), "Negative screen coordinate given") ;

  screen[x][y] = true ;

  ASSERT( screen[x][y] == true, "Failed to light pixel") ;
   return ;
}
```

The same PRE_CONDITION and POST_CONDITION statements could be written as Specification Placed assertion statements (for C++ only), and they would be evaluated identically. Specification Placed assertion statements are written as:

```
class Screen
{
  private:
     bool screen[10][10] ;

  public:
    LightPixel(int x, int y) ;
      PRE_CONDITION((x >= 0) && (y >= 0)), "Negative screen coordinate given") ;
      POST_CONDITION( screen[x][y] = true, "Failed to light pixel") ;
    ClearPixel(int x, int y) ;
}
```

4.3.3 The INVARIANT Statement

An INVARIANT statement is a combination of the PRE_CONDITION and POST_CONDITION statements. It is used to check that a condition holds at both the beginning and end of every public method in a class. It is equivalent to writing both a PRE_CONDITION and POST_CONDITION for each public method. In a constructor or destructor, it acts as only a POST_CONDITION or PRE_CONDITION respectively. An INVARIANT statement must be placed in the specification of a class, and this feature is only available in the C++ capability of ASSERT++. The statement is placed immediately after the opening "curly brace" at the beginning of a class specification. An example of an INVARIANT statement is:

```
class Screen
{
  INVARIANT( (cursorX > 0) && (cursorY > 0), "Negative coordinate given") ;
  private:
     bool screen[10][10] ;
     int cursorX ;
     int cursorY ;

  public:
    LightCursor(void) ;         // Lights up the current cursor position
}
```

This is equivalent to writing:

```
class Screen
{
private:
     bool screen[10][10] ;
     int cursorX ;
     int cursorY ;

  public:
    LightCursor(void) ;         // Lights up the current cursor position
      PRE_CONDITION( (cursorX > 0) && (cursorY > 0),
                     "Negative coordinate given") ;
      POST_CONDITION( (cursorX > 0) && (cursorY > 0),
                     "Negative coordinate given") ;
}
```

4.4 The Firing of Assertions

When the condition of an assertion statement evaluates to false, the assertion is fired. When an assertion is fired, the user is supplied with some useful debugging information. This information can be used to determine exactly which assertion statement fired, and can aid in determining why that statement fired. Each of the assertion types listed above produce a similar output when they are fired. In C++ this output looks like:

```
<process ID>: <assertion type> <condition> failed at <file name>:<line number>
```

The $< processID >$ is the unix process ID of the executable program that contained the assertion that fired. The $< assertiontype >$ tells whether the assertion that fired was a pre-condition, post-condition, assertion, or invariant. The $< condition >$ is the condition that failed, causing the assertion to fire. The $< filename >$ is the name of the file that contains the source code of the location where the assertion fired, and the $< linenumber >$ is the line in this file where the assertion was fired.

The $< filename >$ and $< linenumber >$ outputs of Specification Placed assertions are adjusted to reflect the location in the source code that they were evaluated at. This means that a PRE_CONDITION statement that is Specification Placed will show the file name and line number that identifies the opening "curly brace" of that method's definition. This feature is even more useful when dealing with POST_CONDITION and INVARIANT statements. When a POST_CONDITION fires in a method that has multiple exit points, the file name and line number can be used to determine which exit point was being executed when the assertion fired. Similarly, the firing of an INVARIANT can be traced to discover exactly where execution was when it failed.

The following is an example of a POST_CONDITION firing during the running of an executable. The file "cls.cxx" is a library file that is being used by the compiled program "test_double".

```
cls.cxx:


. . . .


37: /* DoubleInt(int x)
38:  *    This function takes an integer as its parameter, and returns an
39:  *    integer that is twice as large as the integer that it was given.
40:  */
41: int
42: DoubleInt (int x)
43: {
44:   POST_CONDITION ($RETURN == ($OLD(x) * 2), "Failed to double the variable");
45:
46:   x *= 3 ;     // This is where the error occurs!
47:   return x ;
48: }


. . . .
```

When the program is run...

```
rst_computer> test_double
process 11350: post-condition ( $RETURN == ( $OLD ( x ) * 2 )  )
failed at cls.cxx:47
[Failed to double the variable]
```

Notice that the file name indicated ("cls.cxx") is the the name of the library file that contains the DoubleInt function, not the name of the program that was executed. Also notice that the line number that is displayed is the line number of the return statement (47), not the line number of the post-condition. This information makes it simple to track down the bug in this program.

In Java, the message that is displayed is slightly different due to the different capabilities of the language. In Java, no process id is displayed. Java does give the added information of an entire stack trace where the line number and file name are displayed in C++. An example of an assertion firing in Java would look something like:

```
rst_computer> java prime
PostCondition ($ForAll ( int , x , 0 , 5 , q [ x ] > - 1 ) ) failed:
        at prime.arr_test(prime.java:77)
        at prime.main(prime.java:116)
[negative int existed]
```

4.5 Assertion Behaviors

There are a number of actions that can be taken when an assertion fires. These actions, called behaviors, can be determined by the user. Both the C++ and the Java tool support behaviors that allow the user to specify whether to continue, terminate, or ignore when an assertion fires. The C++ tool adds the capability to have the program pause in its execution. The Java tool gives the user the option of having the output from the terminate or continue behavior go to either stdout or to a special pop-up window that the user must close.

The most common behavior that will be used is the terminate behavior. This behavior causes a program to cease execution after an assertion statement has fired and printed its output. Since the firing of an assertion indicates a bug in the code, it is usually desirable to stop execution at this point instead of continuing onward.

The continue option causes the program to continue execution even after an assertion statement has fired. It prints the output that it normally would, and then continues with the program (possibly causing more assertion statements to fire later). This behavior might be useful for things such as tracing the propagation of an error through a program or for collecting the output of a program in a file (possibly for testing purposes).

The ignore behavior causes an evaluated assertion to produce no output. This is different from removing the assertions from the source code (talked about later) because the assertion statement is still evaluated.

The pause behavior (C++ only) results in output being printed as usual, and the program being put into a paused state. This is a useful debugging tool because, by using the process ID that is given, the user can load the paused program into a debugger. Since the program has not yet terminated, the programmer has access to the complete state of the program at the time that the assertion was fired, and the programmer can use this information to determine what caused the firing of the assertion. Each of these behaviors can be configured at either compile time or run time. This enables the user to compile a program once, and run it using various behavior settings. In addition to these behavior settings, there are other options that affect assertion statements. Each of the assertion types may be turned on or off independently. The user may also choose to turn off at compile time. When the assertions are turned off, they are not compiled as part of the source code. This is most useful when a program has been thoroughly tested, and is being made ready for release.

4.6 The Assertion Language

The assertion language that RST developed is used to specify the condition that an assertion statement is verifying. It functions as an extension of the syntax that C++ and Java provide. The assertion language provides a means to simplify the creation of complex boolean expressions. An assertion is fired when one of these boolean expressions evaluates to false. RST's assertion language extends the power of boolean expressions and make them more readable. The following language constructs are provided (and can be joined with any standard C++ / Java expression):

$NOT, $AND, $OR These are just textual equivalents of "!", "&&", and "||" respectively. They help to increase code readability.

$Always Evaluates to false; used to cause an assertion to fire unconditionally. Analogous to (false).

$Zero(**$<$ *variable* $>$)** Evaluates to false if $<$ *variable* $>$ is not zero. Analogous to ($<$ *variable* $>== 0$).

$Equal(**$< var1 >, < var2 >$)** Evaluates to false if $< var1 >$ does not equal $< var2 >$. Analogous to ($< var1 >==< var2 >$).

$Range(**$< variable >, < lower >, < upper >$)** Evaluates to false if $< variable >$ is not between $< lower >$ and $< upper >$. Analogous to (($< variable >$ >= *lower*) && ($< variable >< upper$))

$StringCompare(**$< string1 >, < string2 >$)** The string parameters must both be type char *. Evaluates to false if the strings are not identical.

$ForAll(**$< type >, < iterator >, < initialization >, < stop >, < condition >$)** The $ForAll statement is can be used to ensure that every element of an array or STL container meets some condition. It is based on the mathematical concept of universal quantification. The user must supply the iterator and its type. He must also provide the initialization value for the iterator, and the stopping criteria. The $< condition >$ is checked at each iteration of the loop. The $< condition >$ may be a simple expression, or it may be a complex expression built using any of the elements of the assertion language. This includes the ability to nest $ForAll statements within other $ForAll or $ThereExists (see below) statements. The following is an example of using a $ForAll statement in an assertion statement.

```
. . . .
int grades[10] ;
```

```
. . . .
ReadGrades(grades) ;
ASSERT($ForAll (int, i, 0, 10, grades[i] >= 0),
       "No negative grades allowed") ;
. . . .
```

This assertion statement will fire if any of the integers stored in the array grades (between 0 and 9 inclusive) are negative.

$ThereExists(< *type* **>, <** *iterator* **>, <** *initialization* **>, <** *stop* **>, <** *condition* **>)** The $ThereExists statement follows the same format as the $ForAll statement. It is based on the mathematical concept of existential quantification. It will fire only if the condition fails for every iteration of the loop. Like the $ForAll statement, it too may be nested or combined with any element of the assertion language. The expression:

```
  ThereExists(int, i, 0, 10, grades[i] >= 65)
```

will evaluate to false only if none of the integers stored in grades (between 0 and 9 inclusive) are greater than 65.

$RETURN and $OLD(< *variable* **>)** These two keywords are available for use only in a POST_CONDITIOIN statement. The $RETURN keyword refers to the value being returned by the function at whatever return statement has been reached.
The $OLD(< *variable* >) keyword is used to reference the value of the variable < *variable* > at the entrance to the method / function.
The $OLD and $RETURN keywords can be combined and used as follows:

```
int
DoubleInt (int x)
{
  POST_CONDITION($RETURN == $OLD(x) * 2), "Failed to double the variable") ;
  x *= 2 ;
  return x ;
}
```

In this example, the POST_CONDITION is ensuring that the value that is returned by DoubleInt() is twice that of what was passed in.

5 Using Testability to Guide Assertion Placement

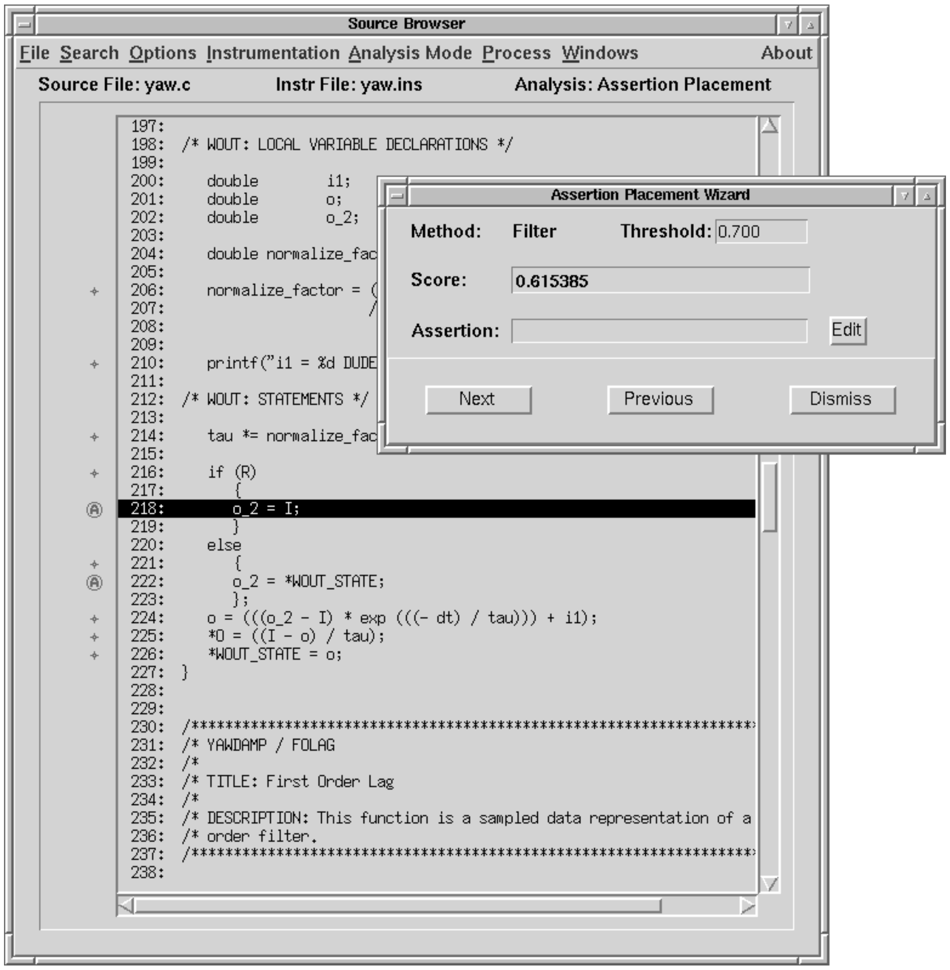
The novelty of this entire SBIR project was the idea of using testability scores (which are derived from a fault injection technique called sensitivity analysis) to suggest where assertions can find faults that a test suite cannot. Testability scores are composed of three factors: execution estimates, infection estimates, and propagation estimates.

After much research, and given our results from Phase I [4], the propagation estimates were determined to be the most crucial information for assertion placement. When you consider that this effort is focused on enhancing the testing process of object-oriented software, this makes intuitive sense. After all, tiny objects are very easily tested in isolation. It is not until they are combined with other tiny objects that we realize that if faults hide, they will hide by virtue of the fact that the corrupted states do not propagate properly. Hence propagation scores provide the most useful information to guide assertion placement.

Propagation scores measure the likelihood that an erroneous data value will result in erroneous program output. The higher the propagation score, the more likely it is that bad output will be produced, and hence it is more likely that regular testing will succeed in finding those types of faults. It is not until low testability scores are observed that concern should occur, since the lower scores indicate that testing is unlikely to find faults at those places. To compute the propagation score for a given program, variables are perturbed and the programs output is checked to see if the output of the perturbed program differs from the output of the unperturbed program.

Testers and developers can usually detect when a program produces corrupted output (if they cannot, the usefulness of testing is seriously brought into question). When programs do "wear their faults on their sleeves" in this manner, the need for additional assertions is reduced. The need for assertions occurs when programs do not have this property, and low propagation scores are observed. Low propagation scores suggest a much greater likelihood that testing will fail to detect existing faults, and thus by adding probes (assertions) into the software before the software is tested, we get a much clearer picture as to whether faults are present.

Examining propagation scores can aid a programmer in the placement of ASSERT statements. The CBrowse utility that we have built as a part of this effort can read propagation results and display them to the user. CBrowse is capable of loading multiple files, and has been modified to help the user place assertion statements. RST has also created an Assertion Placement Wizard that aids the user in stepping through propagation results loaded in CBrowse. The Wizard asks the user to choose an upper bound for testability scores (the threshold). It then allows the user to move between statements that have scores below this threshold by clicking on either the Next or Previous buttons (See Figure 5). The Wizard displays the score for the current statement, and the assertion (if one exists). When all of the relevant scores in the current file have been visited, the tool will automatically load another file from the program and locate the first testability score that is within the specified threshold. The Wizard provides an easy means of navigating through source-code.


Source Browser
File Search Options Instrumentation Analysis Mode Process Windows About
Assertion Placement Wizard
Edit
Next
Previous
Dismiss

Source File: yaw.c   Instr File: yaw.ins   Analysis: Assertion Placement

```

/* WOUT: LOCAL VARIABLE DECLARATIONS */

   double       i1;
   double       o;
   double       o_2;

   double normalize_fac

   normalize_factor = (
                      /


   printf("i1 = %d DUDE

/* WOUT: STATEMENTS */

   tau *= normalize_fac

   if (R)
      {
      o_2 = I;
      }
   else
      {
      o_2 = *WOUT_STATE;
      };
   o = (((o_2 - I) * exp (((- dt) / tau))) + i1);
   *O = ((I - o) / tau);
   *WOUT_STATE = o;
}


/*******************************************************************
/* YAWDAMP / FOLAG
/*
/* TITLE: First Order Lag
/*
/* DESCRIPTION: This function is a sampled data representation of a
/* order filter.
/*******************************************************************

```

Method: Filter   Threshold: 0.700

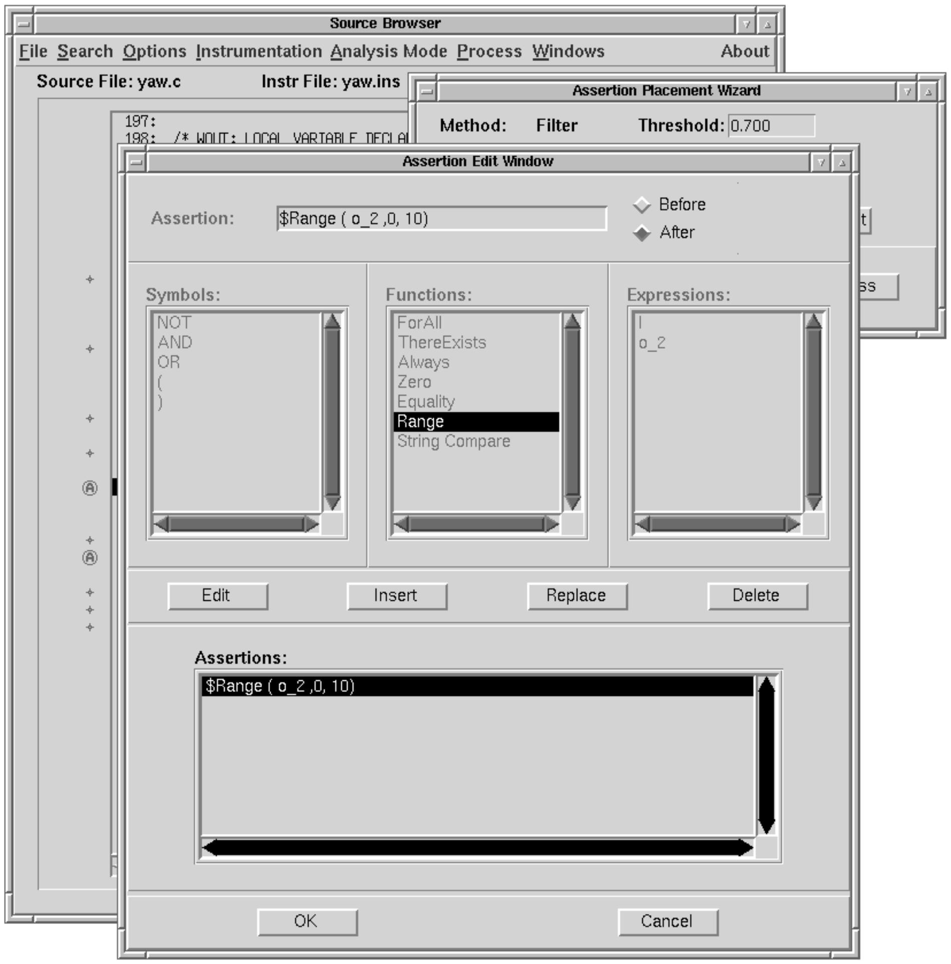
Score: 0.615385

Assertion:

Figure 5: Assert Wizard for Placement.

The Assertion Edit Window (AEW) can be used in conjunction with the Assertion Wizard to simplify the placing of ASSERT statements. The AEW provides a GUI that allows the user to place complex assertions with less effort (See Figure 6).

The user brings up the AEW utility by clicking on the Edit button of the Assertion Placement Wizard. The AEW provides lists of symbols, expressions, and relevant variables that allow the user to build ASSERT statements with minimal typing. When building an assertion statement, the user can use any item in these lists by simply clicking on it. The AEW also provides guidelines to help build syntactically correct statements. After building the assertion statement, the user has the option of placing the ASSERT statement either before or after the statement that is currently being examined. When the user inserts an assertion statement it appears in the Assertions window.



Source File: yaw.c
Instr File: yaw.ins

```
197:
198:  /* WOUT: LOCAL VARIABLE DECLA
```

Method: Filter
Threshold: 0.700

Assertion: $Range (o_2 ,0, 10)

Before [✓ After]

$Range (o_2 ,0, 10)

Figure 6: A.E.W.

When the user places an ASSERT statement using the AEW, it is not immediately placed in the program's source code. Assertions built in this manner are first stored in memory, and then placed in the source code of a program when the user choses the FILE - SAVE ASSERTIONS menu option. Once the user has placed his or her ASSERT statements into the source code by choosing the FILE - SAVE ASSERTIONS menu option, the user must re-instrument for testability (using RST's testability tool) in order to attain valid testability results.

6 Summary

This SBIR Phase II effort provided NIST with a methodology and a tool for predicting where faults will hide during testing and where assertions should be placed. Our deliverable tools work for the C++ and Java languages. To our knowledge, these tools are the only of their kind, and are based on nearly 10 years of basic research into why faults hide during testing. The key US Government organizations that have funded this research are: National Institute of Standards and Technology, NASA, National Science Foundation, and DARPA.

This methodology improves the testability for a fixed test suite. It is true that bigger test suites could be used as an alternative way to boost testability while ignoring assertions altogether. This issue has cost trade-offs that need further exploration to determine whether smaller test suites with assertions are better at revealing faults than larger test suites without assertions.

Generating additional test cases and embedding assertions are not the only verification and validation tricks that could be employed once it is known where testing is unlikely to be capable of detecting faults. Manual inspections or formal proofs that certain semantic constraints are true could also be applied to ensure that defects are not hiding.

Our conclusion that assertions are beneficial to software testing parallels the comments by Osterweil and Clarke concerning the value of assertions to testing; in their 1992 *IEEE Software* article, they classified assertions as "among the most significant ideas by testing and analysis researchers" [3]. From our previous work studying why faults hide during testing, we believe that we have provided insights into why assertions work well and how their placement can be made more systematic and practical.

We conjectured that assertions that are based on deficits in some testing suite D may still be valuable tools for improving the defect observability rates of another testing suite, D'. If generally true, this suggests that current research into which testing approach is better may be wasted effort, i.e., possibly any testing technique can be massaged into an excellent fault detector after assertions are instrumented to test untestable regions. Most research in software testing is geared toward finding some method K that will produce a test suite D for which $T(P)_D \approx 1.0$. This suggests that when high fault detection is the goal, instead of looking for the ultimate K, derive assertions and generate additional tests, Δ, such that $T(P_{A_D})_{(D \cup \Delta)} \approx 1.0$. In summary, assertions and additional test cases (that ensure reachability) can transform your code from being untestable to testable. Let them.

References

[1] H. Yin and J.M. Bieman. Improving software testability with assertion insertion. In *Proc. of International Test Conference*, pages 831–839, October 1994.

[2] M. Blum. Designing Programs To Check Their Work. Technical report, University of California at Berkley, December 1988.

[3] L. Osterweil and L. Clarke. A Proposed Testing and Analysis Research Initiative. *IEEE Software*, pages 89–96, September 1992.

[4] Reliable Software Technologies Corporation. Testability of Object-Oriented Systems. Technical report, June 1995. NIST Report GCR 95-675, National Institute of Standards and Technology.

[5] D. Luckham and F. von Henke. An overview of ANNA, a specification language for Ada. *IEEE Software*, pages 9–22, March 1985.

[6] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM*, October 1969.

[7] B. Meyer. *Eiffel the Language.* Prentice-Hall, 1992.

[8] J. Voas and K. Miller. Software Testability: The New Verification. *IEEE Software*, 12(3):17–28, May 1995.

[9] P. Naur. Proof of algorithms by general snapshots. *BIT*, 6(4):310–316, 1966.

[10] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[11] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th International Conference on Software Engineering*, May 1992.

[12] D. Rosenblum. Towards a method of programming with assertions. In *Proceedings of the 14th International Conference on Software Engineering*, pages 92–104, May 1992.

[13] R. Rubinfeld. A mathematical theory of self-checking, self-testing, and self-correcting programs. Technical Report TR-90-054, Int. Computer Science Institute, October 1990.

[14] E. J. Weyuker. Axiomatizing software test data adequacy. *IEEE Trans. on Software Engineering*, 12(12):1128–1137, December 1986.

[15] J.M. Bieman and H. Yin. Designing for software testability using automated oracles. In *Proc. of International Test Conference*, pages 900–907, September 1992.